Python 2 から Python 3 への移植

リリース *3.10.18*

Guido van Rossum
and the Python development team

7 月 08, 2025

目次

author Brett Cannon

概要

現在は Python 3 が最新版の Python ですが、Python 2 もまだ活発に利用されています。なのであなたのプロジェクトを両方のメジャーリリースにおいて動作可能にしておくのがよいでしょう。このガイドでは、Python 2 と 3 を同時にサポートするにはどうすればよいかを解説します。

もしあなたが標準 Python ライブラリではなく拡張ライブラリでの移植手段を探しているならば cporting-howto を参照してください。

コア開発者の視点から Python3 が世に出てきたが理由を読みたい場合は、Nick Coghlan の Python 3 Q & A または Brett Cannon による 'Why Python 3 exists がおすすめです。

For help with porting, you can view the archived python-porting mailing list.

1 短い説明

あなたのプロジェクトを、単一ソースで Python 2/3 両方に対応させる基本的なステップは次のとおりです。

1. Python 2.7 だけをサポートすることに気を配ってください。
2. Make sure you have good test coverage (coverage.py can help; `python -m pip install coverage`)
3. Python 2 と 3 の違いを学びましょう。
4. Use Futurize (or Modernize) to update your code (e.g. `python -m pip install future`)
5. Use Pylint to help make sure you don't regress on your Python 3 support (`python -m pip install pylint`)
6. Use caniusepython3 to find out which of your dependencies are blocking your use of Python 3 (`python -m pip install caniusepython3`)
7. Once your dependencies are no longer blocking you, use continuous integration to make sure you stay compatible with Python 2 & 3 (tox can help test against multiple versions of Python; `python -m pip install tox`)
8. Consider using optional static type checking to make sure your type usage works in both Python 2 & 3 (e.g. use mypy to check your typing under both Python 2 & Python 3; `python -m pip install mypy`).

注釈: Note: Using `python -m pip install` guarantees that the `pip` you invoke is the one installed for the Python currently in use, whether it be a system-wide `pip` or one installed within a virtual environment.

---

2 詳細

Python 2 と 3 の同時サポートについてのキーポイントのひとつは、**今日から** 開始出来る、というものです。たとえあなたが持っている依存物がまだ Python 3 をサポートしていなくとも、それはあなたのコードを Python 3 サポートのために **今すぐ** 現代化出来ないことを意味するのではありません。Python 3 サポートのために必要なほとんどの変更は、Python 2 コード内にあっても新しいプラクティスを伴う明快なコードに導いてくれます。

もうひとつのキーポイントは、あなたの Python 2 コードの Python 3 サポートを加える現代化は、あなたのために大部分は自動化されているということです。Python 3 によるテキストデータとバイナリデータの明確な区別のおかげで、あなたはいくつかの API に決断をしなければならないかもしれない一方で、下位レベルの仕事は今やほとんど済んでいて、それゆえに最低でもその自動化された修正からの恩恵をすぐさま受けることが出来ます。

Python 2 と 3 の同時サポートのために、あなたのコードを移植するための以降の詳細を読む際には、これらのキーポイントを心に留めておいてください。

2.1 Python 2.6 とそれ以前のサポートを落とす

While you can make Python 2.5 work with Python 3, it is **much** easier if you only have to work with Python 2.7. If dropping Python 2.5 is not an option then the six project can help you support Python 2.5 & 3 simultaneously (`python -m pip install six`). Do realize, though, that nearly all the projects listed in this HOWTO will not be available to you.

Python 2.5 以下のサポートをスキップ出来るならば、あなたのコードに必要な変更は Python の常套句のような外観と雰囲気を壊すべきではありません。最悪の場合あるインスタンス内のメソッドの代わりに関数を使う必要があったり、ビルトインを使う代わりに関数をインポートする必要があるでしょうが、そうしないならば、全体通した変換はあなたにとって異質に感じさせないものに違いありません。

ですが、Python 2.6 以上と言わず Python 2.7 を目標にしてください。Python 2.6 はもう積極的にはサポートされていません。これは **あなたが** Python 2.6 に関係するあらゆる問題に取り組まなければならないことを意味します。この HOWTO で言及しているいくつかのツールも Python 2.6 をサポートしていません (Pylint など) し、時につれこのようなことはもっと当たり前になってくるでしょう。2.7 以上だけをサポートするということは、話をより簡単にしてくれます。

2.2 あなたの `setup.py` ファイルに、相応しいサポートバージョンを明記することを忘れないこと

`setup.py` ファイルに、あなたがサポートする Python バージョンを Trove 分類 で正しく明記すべきです。あなたのプロジェクトはまだ Python 3 をサポートしていないので、少なくとも `Programming Language :: Python :: 2 :: Only` と明記すべきです。理想的には Python のメジャー/マイナーバージョンも指定すべきです。例えば `Programming Language :: Python :: 2.7` のように。

2.3 良いテストカバレッジを確保する。

そうしたい一番古いバージョンの Python 2 をサポート出来ているならば、あなたのテストスイートが十分な網羅性かを確認したいでしょう。あなたのコードをツールで書き換えた後に現れるあらゆる失敗が実際にはツールのバグで、あなたのコードのバグではないとするのに十分なだけの確信をあなたのテストスイートに持ちたいならば、良い経験則がこれです。目標とする数値で言えば、80% 以上の網羅性を目指してみてください (そしてカバレッジ 90% を越えるのが難しかったとしても気に病む必要はありません)。テストカバレッジの計測ツールを手持ちでないならば、coverage.py がお奨めです。

2.4 Python 2 と 3 の違いを学びましょう。

Once you have your code well-tested you are ready to begin porting your code to Python 3! But to fully understand how your code is going to change and what you want to look out for while you code, you will want to learn what changes Python 3 makes in terms of Python 2. Typically the two best ways of doing that is reading the "What's New" doc for each release of Python 3 and the Porting to Python 3 book (which is free online). There is also a handy cheat sheet from the Python-Future project.

2.5 コードをアップデートする。

Python 2 と比較した Python 3 の違いがわかってきたら、いよいよあなたのコードを更新するそのときです! あなたのコードの移植の自動化ツールとしては 2 つの選択肢があります: Futurize と Modernize です。どちらのツールが良いかはあなたのコードをどのくらい Python 3 寄りに近付けたいかによります。Futurize は、例えば Python のメジャーバージョン間の意味論的な等価性を持つように Python 3 からバックポートされた `bytes` 型のように、Python 2 に取り込まれた Python 3 のイディオムと慣例を積極的に使います。他方 Modernize はより保守的で、互換性保持を six によって提供することで、Python 2/3 のサブセットであることを目標にします。Python 3 は確実にやってくる未来なので、Python 3 で導入された、まだ慣れていない新しい慣例に合わせ始めるためには Futurize を検討するのが最良かもしれません。

どちらのツールを選ぶにせよ、それらはあなたのコードを、あなたが開始した Python 2 バージョンへの互換性を保ったままで Python 3 で動作するように書き換えます。念には念を入れたければ、まずはテストスイートに対してツールを適用して、変換が正しいものであることを確認するために差分を視覚的に点検しましょう。あなたのテストスイートを変換して、テストがそれでもまだ期待通りにパスすることが検証出来てしまえば、あなたのアプリケーションコードを、全ての失敗するテストは変換の失敗を意味することがわかる状態で変換出来ます。

悪い報せ。これらツールは Python 3 であなたのコードを動作させるために、全ての自動化が出来ているわけではありませんので、Python 3 のフルサポートのためには手動で更新しなければならないわずかばかりの事項があります (必要な手作業はツールによって違います)。選んだツールのドキュメントを読んで、デフォルトでは何が修正されて、選択的に何を修正する (しない) を選べるのか、そして何を自身で修正する必要があるのかを理解してください (例えばビルトインの `open()` ではなく `io.open()` を使う修正は、Modernize ではデフォルトでオフです)。良い報せ。ですが、注意深くみなければデバッグを困難にするような大きな問題として考えられる、警戒するようなことは、2 つだけです。

除算

Python 3 では、`5 / 2 == 2.5` であり `2` ではありません; `int` 同士の全ての除算は `float` の結果になります。この変更については実際のところ、2002 年にリリースされた Python 2.2 から計画されました。そのようなわけで、`/`, `//` 演算子を使うどんなファイルにも `from __future__ import division` を追加するか、あるいはインタプリタを `-Q` フラグとともに起動することが推奨されていました。これをまだやったことがなければ、コードをくまなく調べて対象箇所を見つけ、2 つのことをします:

1. `from __future__ import division` をあなたのファイルに追加します
2. floor division (訳注: float での結果に `floor()` 適用したのと同じ振る舞いをする除算) に対しては `//`

を、浮動小数点数の演算を期待する箇所ではそのまま `/` を使うように、除算演算子を必要に応じて変更します。

オブジェクトが自身の `__truediv__` メソッドを持っているのに `__floordiv__` を持っていない場合に壊れてしまうので、`/` を `//` に単純に自動的に変換することは出来ません (例えばユーザ定義クラスで `/` を何かの演算に使っていて、`//` は同じ事をしないか何もしないような場合)。

テキスト対バイナリデータ

Python 2 では `str` 型をテキストとバイナリデータのどちらにも使うことが出来ていました。不幸なことにこれは、2 つの異なる概念を重ね合わせていて、両方の種類のデータに対して、時々動作して時々はそうではない、といった傷つきやすいコードに繋がりやすいものでした。人々が特定の一つの型の代わりに `str` を受け付ける何かが、それが許容するのはテキストなのかバイナリデータなのかを名言しないときの、悩ましい API を生み出してしまう要因でもありました。これはとりわけマルチリンガルをサポートするための状況を、テキストデータをサポートしていると主張しているのに明示的に `unicode` をサポートすることに注意を払わない API、という形で複雑にしていました。

テキストとバイナリデータの区別をより明快に、よりはっきり宣言するために、Python 3 はインターネット時代に作られたほとんどの言語がしたこと、すなわちテキストとバイナリデータを区別できる別々の型とし、無分別にお互い混ぜこぜには出来ないようにしました (Python はインターネットが広く普及する前からありました)。テキストのみを取り扱うコード、バイナリデータのみを扱うコードのいずれにとっても、この分離は問題を引き起こしません。ですが両方を処理するコードにとっては、それはテキストとバイナリデータの比較をする際に新たな注意点が増えたことを意味していて、これが完全には移行の自動化が出来ない理由なのです。

To start, you will need to decide which APIs take text and which take binary (it is **highly** recommended you don't design APIs that can take both due to the difficulty of keeping the code working; as stated earlier it is difficult to do well). In Python 2 this means making sure the APIs that take text can work with `unicode` and those that work with binary data work with the `bytes` type from Python 3 (which is a subset of `str` in Python 2 and acts as an alias for `bytes` type in Python 2). Usually the biggest issue is realizing which methods exist on which types in Python 2 & 3 simultaneously (for text that's `unicode` in Python 2 and `str` in Python 3, for binary that's `str`/`bytes` in Python 2 and `bytes` in Python 3). The following table lists the **unique** methods of each data type across Python 2 & 3 (e.g., the `decode()` method is usable on the equivalent binary data type in either Python 2 or 3, but it can't be used by the textual data type consistently between Python 2 and 3 because `str` in Python 3 doesn't have the method). Do note that as of Python 3.5 the `__mod__` method was added to the bytes type.

テキストデータ	バイナリデータ
	decode
encode	
format	
isdecimal	
isnumeric	

処理の区別を簡単にするには、バイナリデータとテキストの間のエンコードとデコードを、あなたのコードの

境界で行うようにすることです。バイナリデータとしてテキストを受け取ったならば、即座にデコード。テキストをバイナリデータにして送信する必要があったら、出来るだけあとでエンコード。このようにすることで、あなたのコードは内部的にはテキストだけで動作し、ですから、今処理しているのがどの型なのかを逐一追跡しなくても良くなります。

The next issue is making sure you know whether the string literals in your code represent text or binary data. You should add a `b` prefix to any literal that presents binary data. For text you should add a `u` prefix to the text literal. (there is a `__future__` import to force all unspecified literals to be Unicode, but usage has shown it isn't as effective as adding a `b` or `u` prefix to all literals explicitly)

As part of this dichotomy you also need to be careful about opening files. Unless you have been working on Windows, there is a chance you have not always bothered to add the `b` mode when opening a binary file (e.g., `rb` for binary reading). Under Python 3, binary files and text files are clearly distinct and mutually incompatible; see the `io` module for details. Therefore, you **must** make a decision of whether a file will be used for binary access (allowing binary data to be read and/or written) or textual access (allowing text data to be read and/or written). You should also use `io.open()` for opening files instead of the built-in `open()` function as the `io` module is consistent from Python 2 to 3 while the built-in `open()` function is not (in Python 3 it's actually `io.open()`). Do not bother with the outdated practice of using `codecs.open()` as that's only necessary for keeping compatibility with Python 2.5.

`str` と `bytes` の両方のコンストラクタは同じ引数を与えても Python 2 と 3 で異なる意味を持ちます。Python 2 で `bytes` に数値を与えると、整数の文字列表現を生成します: `bytes(3) == '3'` 。ですが Python 3 では、`bytes` に整数を与えると、整数値で与えたぶんの長さの、null バイトで埋められたバイト列を生成します: `bytes(3) == b'\x00\x00\x00'` 。似たような話はバイト列オブジェクトを `str` に与える場合にも起こります。Python 2 ではバイト列が渡したものがそのまま戻ってきます: `str(b'3') == b'3'` 。対して Python 3 では、バイト列オブジェクトの文字列表現になって返ってきます: `str(b'3') == "b'3'"` 。

最後に、バイナリデータに対するインデクシングには取り扱いに注意が必要です (スライシングには特別な取り扱いは **不要** です)。Python 2 では、`b'123'[1] == b'2'` ですが、Python 3 では `b'123'[1] == 50` です。バイナリデータはただのバイナリ数値の羅列ですから、Python 3 では指示した位置のバイトの整数値を返します。ですが Python 2 の場合、`bytes == str` であるために、インデクシングは bytes の要素一つを取り出すスライスとして振舞います。six プロジェクトには `six.indexbytes()` と名付けられた関数があって、これは Python 3 がそうするように整数値を返します: `six.indexbytes(b'123', 1)` 。

まとめると、以下のようになります:

1. どの API がテキストデータを受付け、どの API がバイナリデータを受け付けるのかを決めてください。
2. あなたのコードが Python 2 で確実に、テキストで動くものは `unicode` でも動くように、バイナリデータで動くものは `bytes` でも動くようにしてください (どのメソッドがそれぞれの型で使えないのかを示した上記テーブルをみてください)。
3. Mark all binary literals with a `b` prefix, textual literals with a `u` prefix
4. バイナリデータをテキストにデコードするのは出来るだけ早く、テキストデータをバイナリデータにエンコードするのは出来るだけ遅く。

5. ファイルは `io.open()` を使って開き、そうすべきときには必ず `b` モードを指定してください。

6. Be careful when indexing into binary data

バージョン検出ではなく機能検出を使う

Inevitably you will have code that has to choose what to do based on what version of Python is running. The best way to do this is with feature detection of whether the version of Python you're running under supports what you need. If for some reason that doesn't work then you should make the version check be against Python 2 and not Python 3. To help explain this, let's look at an example.

Let's pretend that you need access to a feature of `importlib` that is available in Python's standard library since Python 3.3 and available for Python 2 through importlib2 on PyPI. You might be tempted to write code to access e.g. the `importlib.abc` module by doing the following:

```
import sys

if sys.version_info[0] == 3:
    from importlib import abc
else:
    from importlib2 import abc
```

このコードの問題は、Python 4 が出たときに起きます。Python 3 ではなく Python 2 を例外的なケースとして扱い、将来の Python のバージョンは Python 2 よりも Python 3 と互換性があると仮定する方が良さそうです:

```
import sys

if sys.version_info[0] > 2:
    from importlib import abc
else:
    from importlib2 import abc
```

ところが、最適解はバージョン検出を一切せずに、代わりに機能検出に頼ることです。機能検出を使うことで、バージョン検出が上手く行かなくなる潜在的な問題を避けられ、機能の互換性を保つ助けになります:

```
try:
    from importlib import abc
except ImportError:
    from importlib2 import abc
```

2.6 互換性オプション

あなたのコードを完全に Python 3 互換に変換できたら、今度は Python 3 での動作が退化したり止まってしまうことがないようにしたいでしょう。この時点ではまだ実際に Python 3 で動作させられない阻害要因となる依存物を持っている場合に、これは特に当てはまります。

互換性を保ち続けるために、あなたが作る全ての新しいモジュールは、最低でもソースコードの先頭に以下のコードブロックを持つべきです:

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
```

実行時に種々の互換性問題を警告してもらうために Python 2 を `-3` フラグ付きで実行することも出来ます。`-Werror` にすれば警告ではなくエラーになるので、うっかり警告を見逃すことがなくなります。

Pylint プロジェクトとその `--py3k` フラグを使って、Python 3 互換性から乖離し始めている際の警告を受け取ることも出来ます。これにより、Modernize や Futurize を普通に実行してみて互換性を失っていないかを確認する、という必要がなくなります。この場合 Python 2.7 と Python 3.4 以上だけのサポートにすることが必要になります。それが Pylint がサポートする最小の Python バージョンだからです。

2.7 どの依存性があなたの移行を阻んでいるのかチェックする

After you have made your code compatible with Python 3 you should begin to care about whether your dependencies have also been ported. The caniusepython3 project was created to help you determine which projects -- directly or indirectly -- are blocking you from supporting Python 3. There is both a command-line tool as well as a web interface at https://caniusepython3.com.

このプロジェクトは同時にあなたのテストスイートに組み込むことが出来る、もう Python 3 使用を妨げる依存物がなくなった時点で失敗するテストコードも提供しています。これにより、Python 3 での動作を開始する際に、依存物を手動でチェックすることなく即座に気付くことが出来ます。

2.8 あなたの `setup.py` ファイルを更新して Python 3 互換を謳う

あなたのコードが Python 3 で動作するようになったら、`setup.py` の classifiers を `Programming Language :: Python :: 3` を含めるように更新して、Python 2 だけのサポートではないことを明記すべきです。これによって、あなたのコードを利用する人はあなたが Python 2 と 3 をサポートすることを知ることが出来ます。理想的には、今サポートしている Python のメジャー/マイナーバージョンも classifiers に追加したいでしょう。

2.9 継続的インテグレーションを使って互換性を維持し続ける。

Python 3 で完全に動作出来てしまったら、あなたのコードが Python 2、3 の両方でいつでも動くことを保障したいでしょう。おそらく、複数バージョンの Python インタプリタでテストを実施するのに最良のツールは、tox です。継続的インテグレーションシステムに tox を統合して、うっかり Python 2 か 3 のサポートを壊してしまわないようにすることが出来ます。

Python 3 インタプリタで `-bb` フラグを使って、bytes と string 、もしくは bytes と int を比較したときに例外を引き起こしたいと思うでしょう (後者は Python 3.5 から使えます)。デフォルトでは型の異なる比較は単純に `False` を返しますが、テキスト/バイナリデータ処理の分離を誤ったり、バイト列への添え字操作を誤ると、簡単には間違いを見つけられません。このフラグはそれが起こった場合に例外を起こすことで、その種のケースを追跡する助けになります。

そしてこれでほぼ全てです! 今の時点であなたのコードベースは Python 2 と 3 の両方に対して同時に互換です。あなたのテストは、開発時点ではどちらのバージョンでテストすることが多いのかによらずに、誤って Python 2 か 3 の互換性を破壊してしまわないようにも組み立てられるでしょう。

2.10 Consider using optional static type checking

Another way to help port your code is to use a static type checker like mypy or pytype on your code. These tools can be used to analyze your code as if it's being run under Python 2, then you can run the tool a second time as if your code is running under Python 3. By running a static type checker twice like this you can discover if you're e.g. misusing binary data type in one version of Python compared to another. If you add optional type hints to your code you can also explicitly state whether your APIs use textual or binary data, helping to make sure everything functions as expected in both versions of Python.